ECSE 321 Introduction to Software Engineering

Hands-on Tutorials

Table of Contents

- HTML version
- PDF version

1. Java Swing

This tutorial discusses step-by-step the test-driven development of a Java Swing application. It provides a narrative description of the videos with source code extracts.

1.1. Description of Event Registration Desktop Application

Typically, this description is elicited from stakeholders (e.g., potential customers). However, for our purposes, we will assume that it is provided as follows:

- The Event Registration application shall provide the ability to add a participant by specifying the participant's name.
- The Event Registration application shall provide the ability to add an event by specifying the event's name, date, start time, and end time.
- The Event Registration application shall provide the ability to register a participant to an event.

1.2. Domain Modeling: Umple

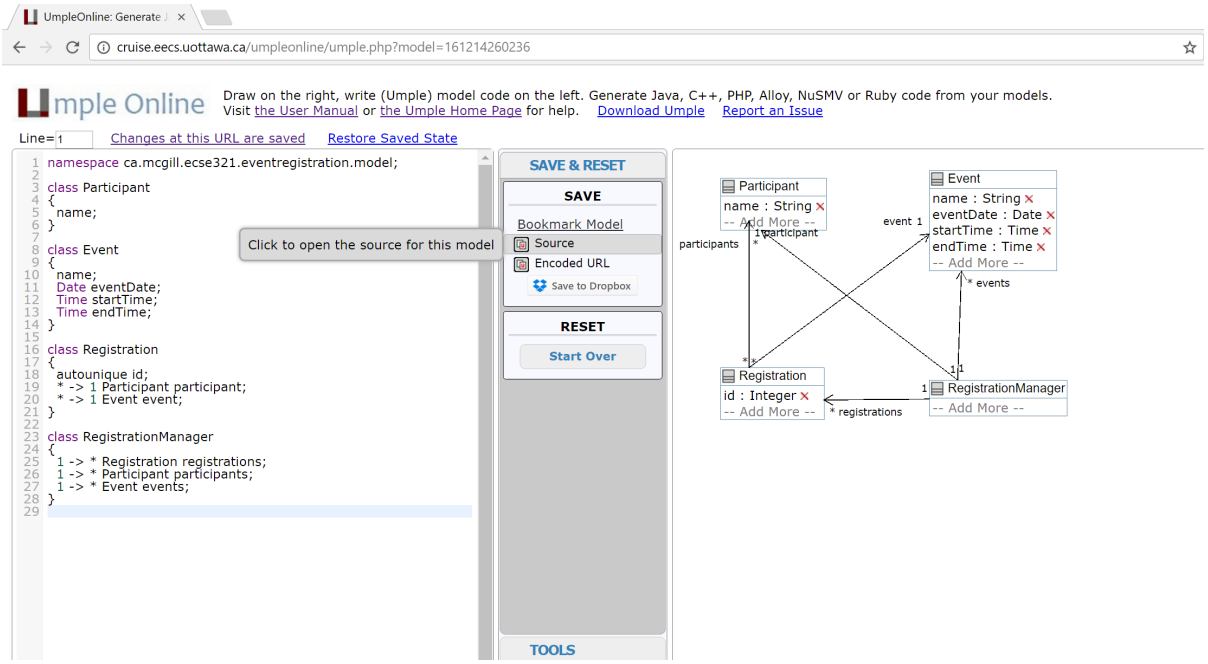
1.2.1. Modeling with Umple online

Umple Online is useful to visualize the domain model with the textual specification at the same time. However if the server is overloaded, simply switch right away to the Umple Eclipse plugin and enter the textual specification there.

1. Ensure that you installed Eclipse Modeling Tools, Neon version. You can check that at **Help** | **About Eclipse**
2. Navigate to http://cruise.eecs.uottawa.ca/umpleonline/ in your browser
3. Add the following piece of Umple code to the left pane

```
namespace ca.mcgill.ecse321.eventregistration.model;

class Participant
{
  name;
}

class Event
{
  name;
  Date eventDate;
  Time startTime;
  Time endTime;
}

class Registration
{
  autounique id;
  * -> 1 Participant participant;
  * -> 1 Event event;
}

class RegistrationManager
{
  1 -> * Registration registrations;
  1 -> * Participant participants;
  1 -> * Event events;
}
```

4. You can arrange the graphical layout of your model in the right pane



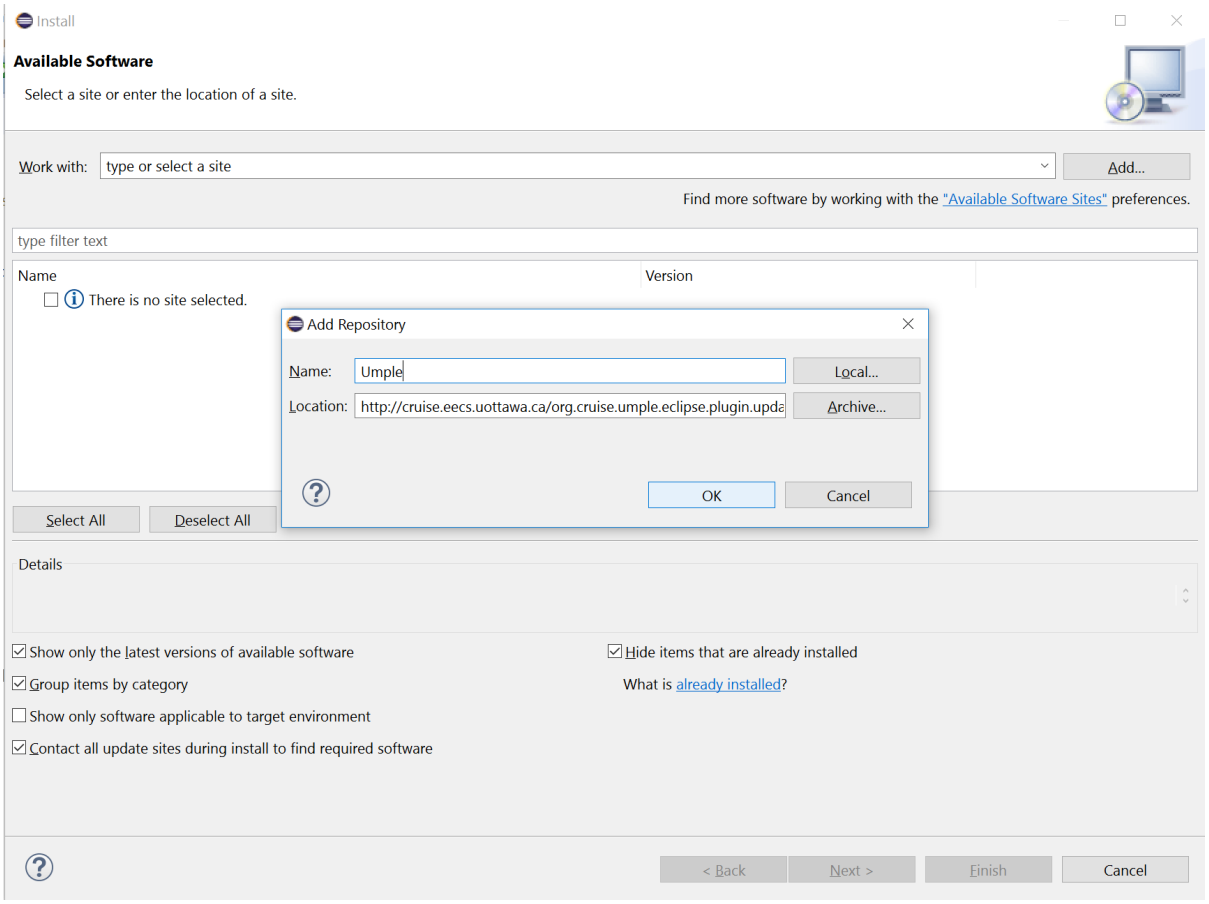
5. Open the *Save & Reset* pane in the middle of the screen
6. Click on *Source* and save your model e.g. as `EventRegistration.ump`

1.2.2. Installing Umple in Eclipse

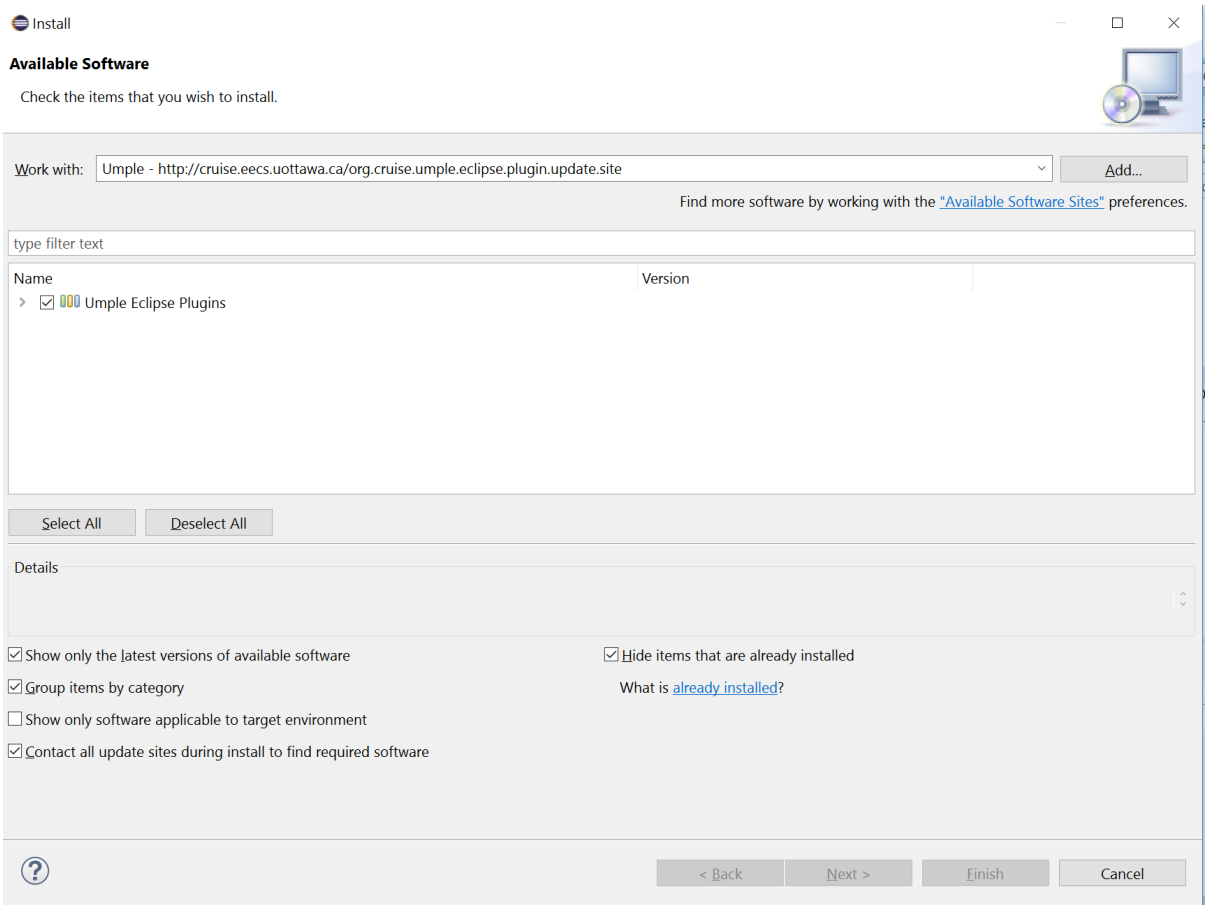
1. Go to **Help | Install new software ...** and click on **Add...**
2. Give **Umple** as name and add the following **Location**:

   http://cruise.eecs.uottawa.ca/org.cruise.umple.eclipse.plugin.update.site

3. Click **OK**

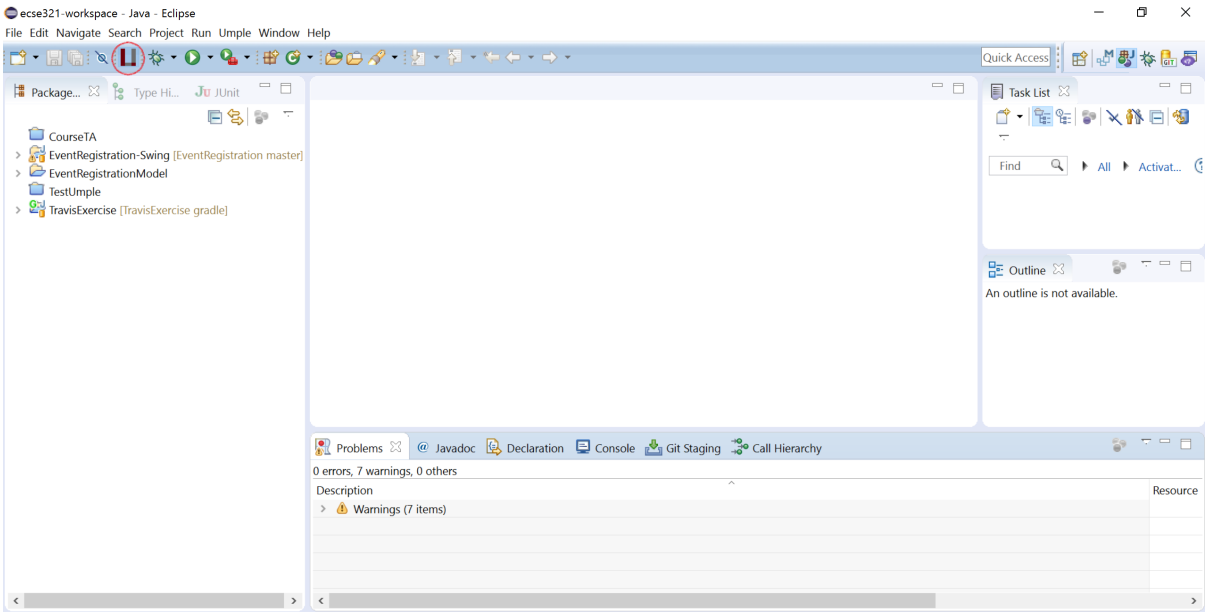


4. Select the checkbox for **Umple Eclipse Plugins** and click **Finish** in the bottom of the screen to install Umple.



5. In case of successful installation, a new **Umple** menu and an icon should appear (see the red

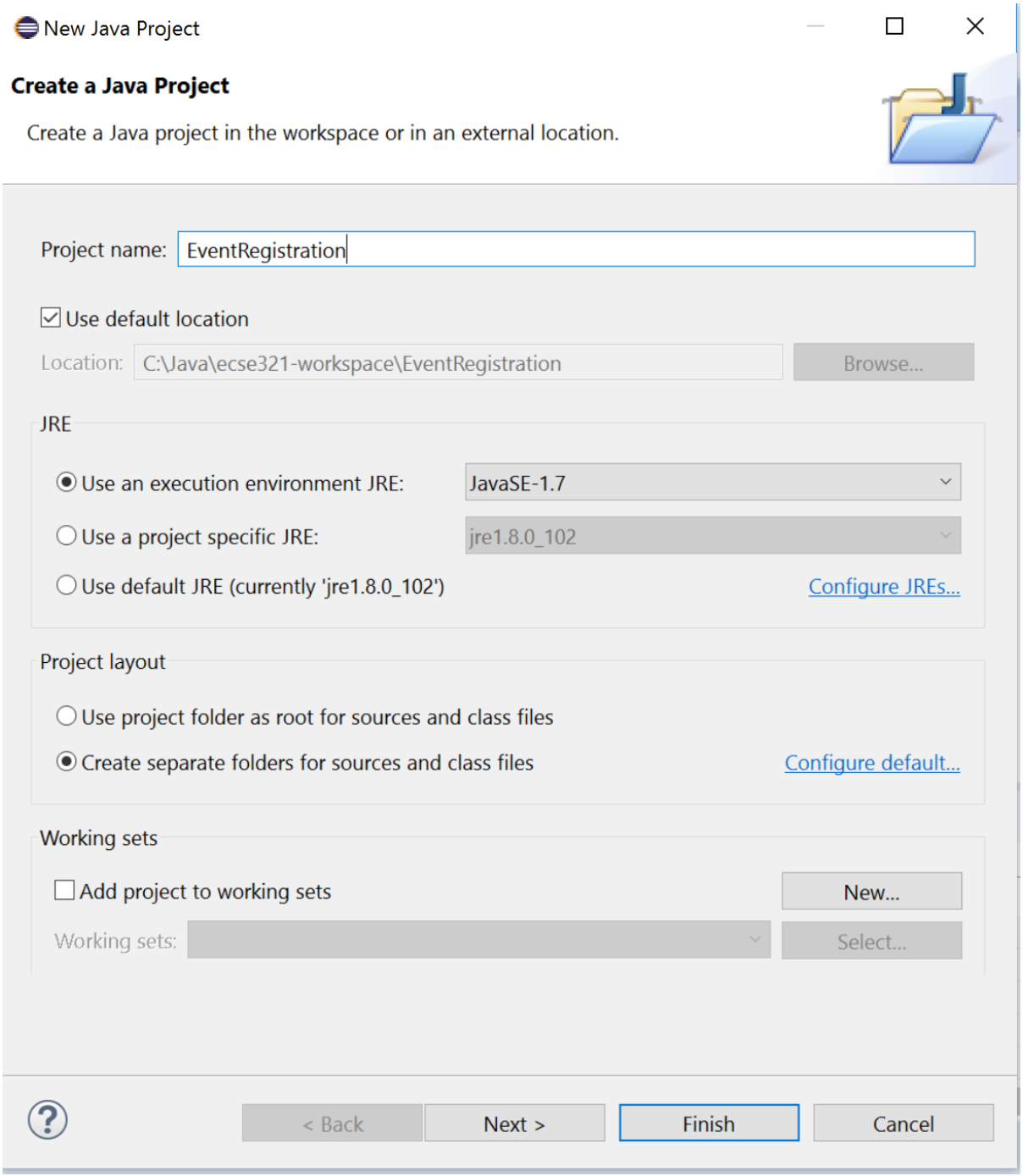
circle).



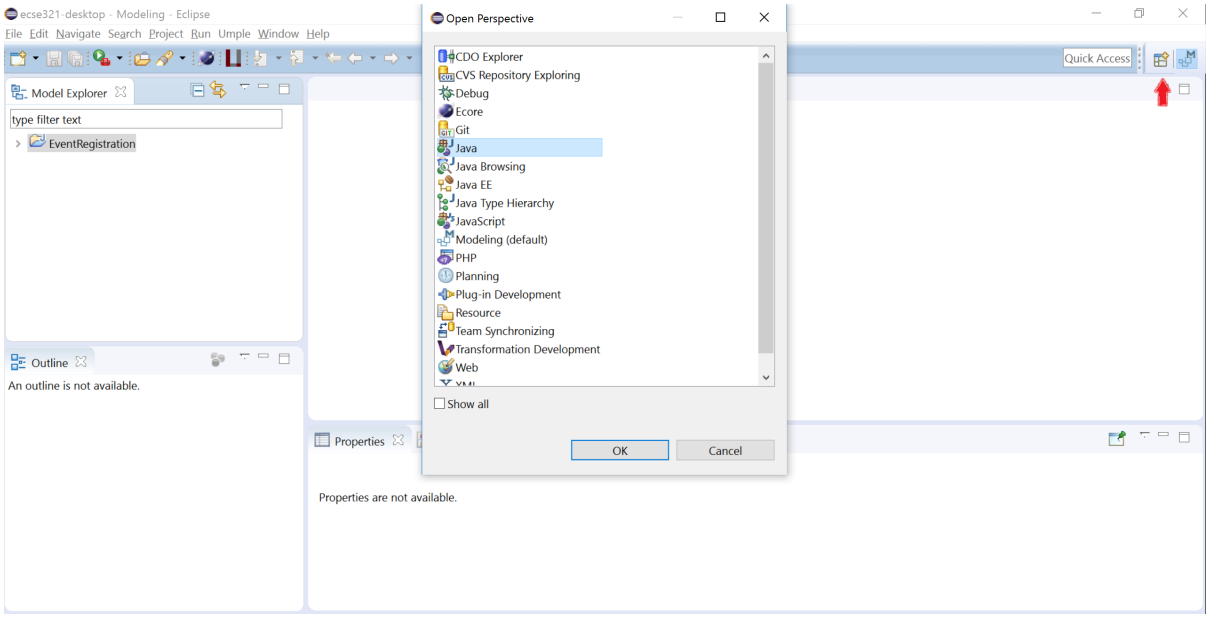
Alternative ways of installing Umple are provided at https://github.com/umple/umple/wiki/InstallEclipsePlugin

1.2.3. Create Java project and add Umple in Eclipse

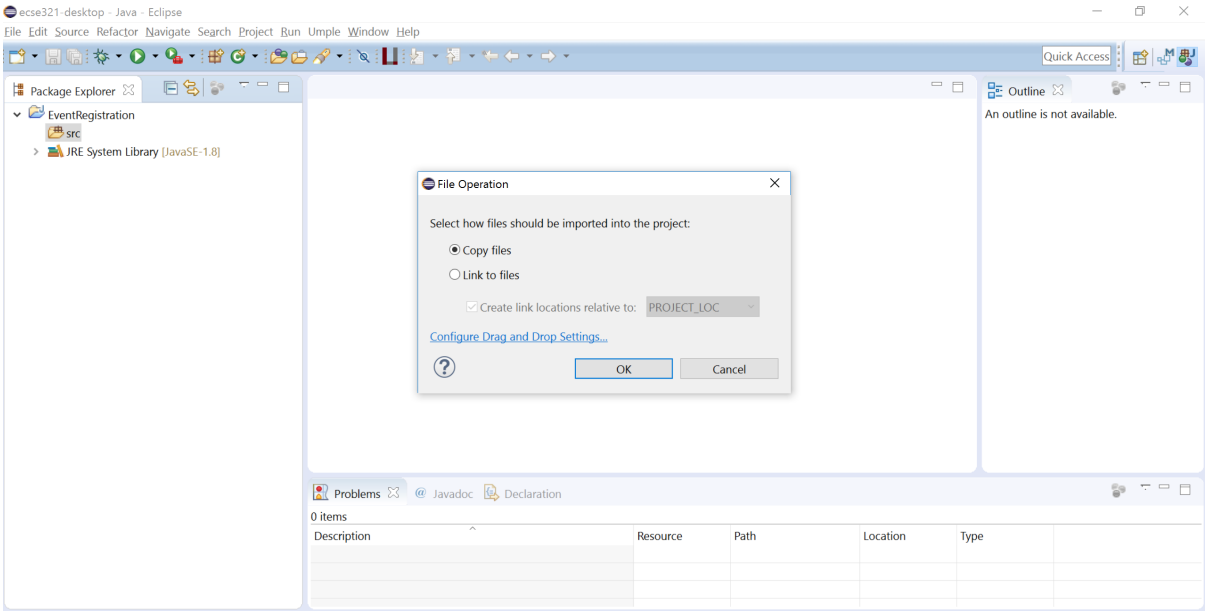
1. Create a new *Java* project by **File** | **New** | **Java Project**
2. Name it as *EventRegistration* and click **Finish**



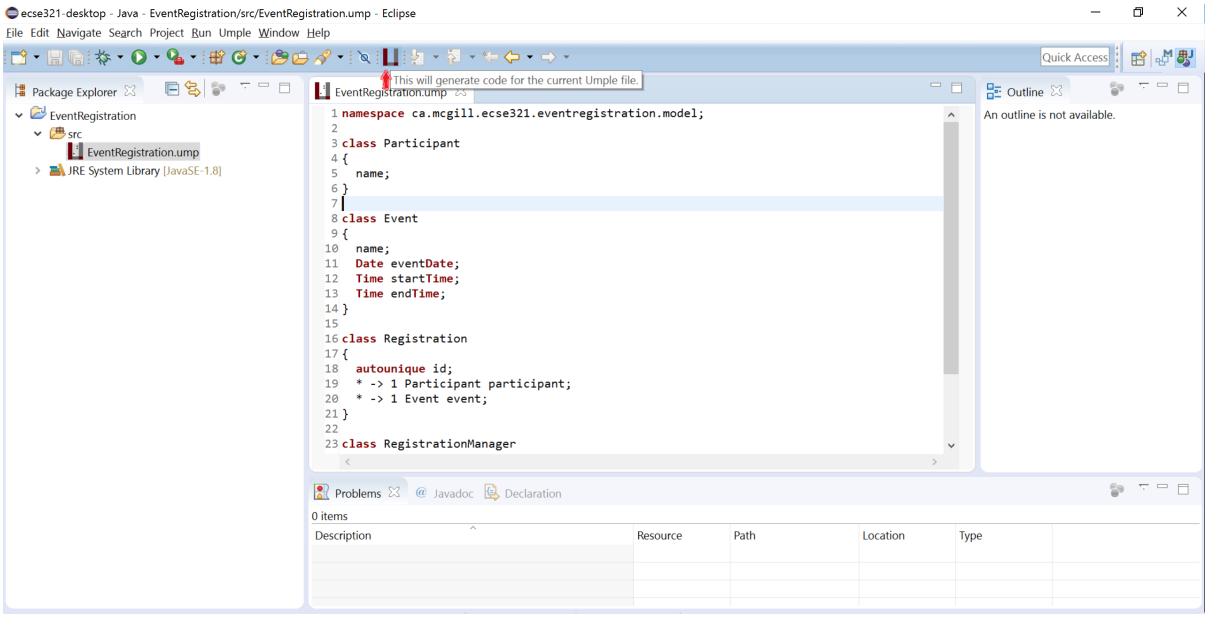
3. Open the *Java perspective* (if not yet open)
    - click on the **Open Perspective** button (see red arrow)
    - then select **Java** and click **OK**



4. **Important**: Due to compatibility reasons, you need to configure Java 1.7 (and not Java 1.8) as a compiler for the project. You can check the project-specific JRE by right clicking on the project and **Build path | Configure build path**
    - Remove the JRE System Library if Java 1.8 is used.
    - Click on **Add Library** and **JRE System Library** then select the appropriate JRE.
5. Drag and drop the Umple model file to the `src` folder and select **Copy files** as option

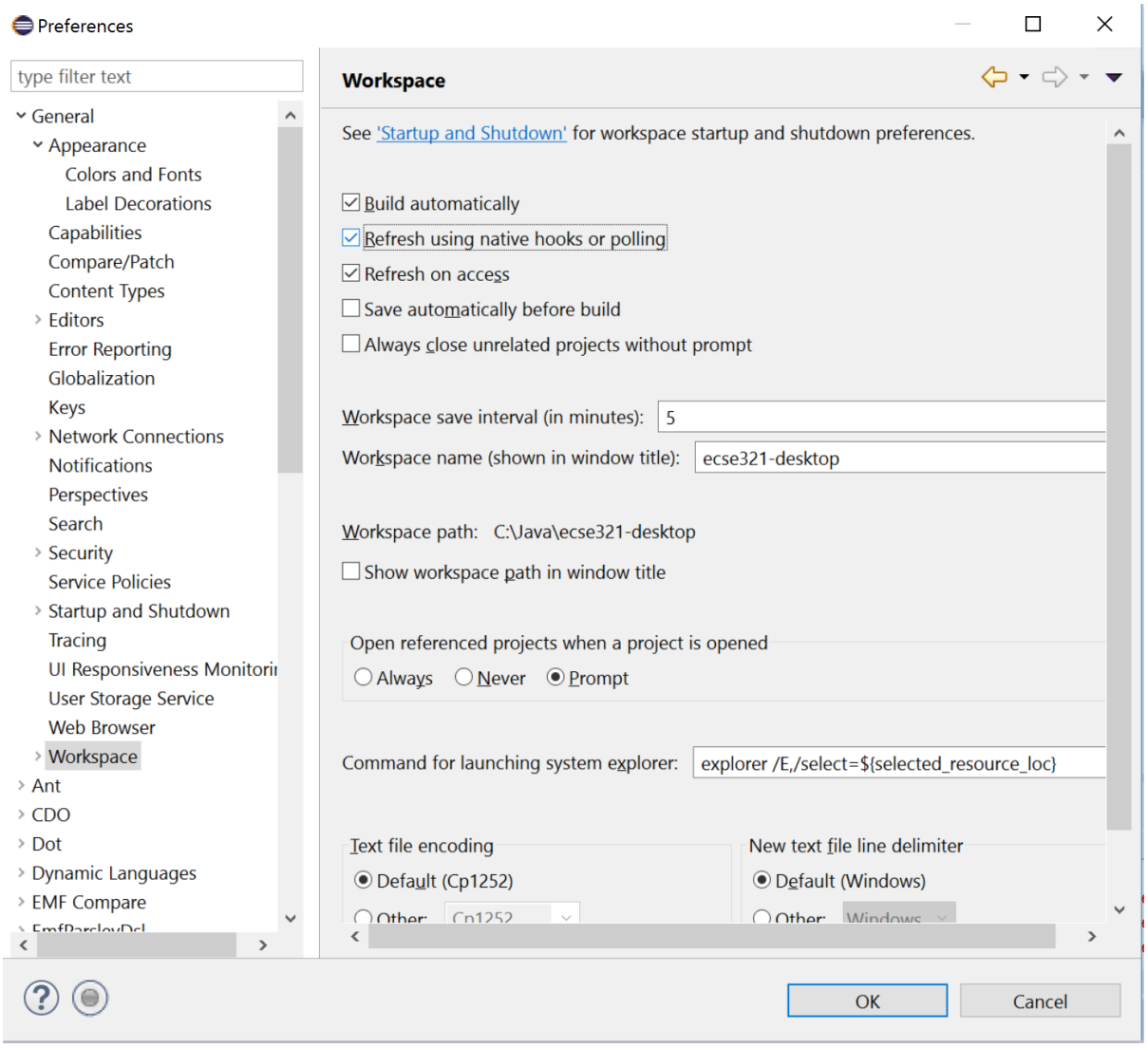


6. Open the Umple file by double click on `EventRegistration.ump`
7. Generate Java code by clicking on the Umple button

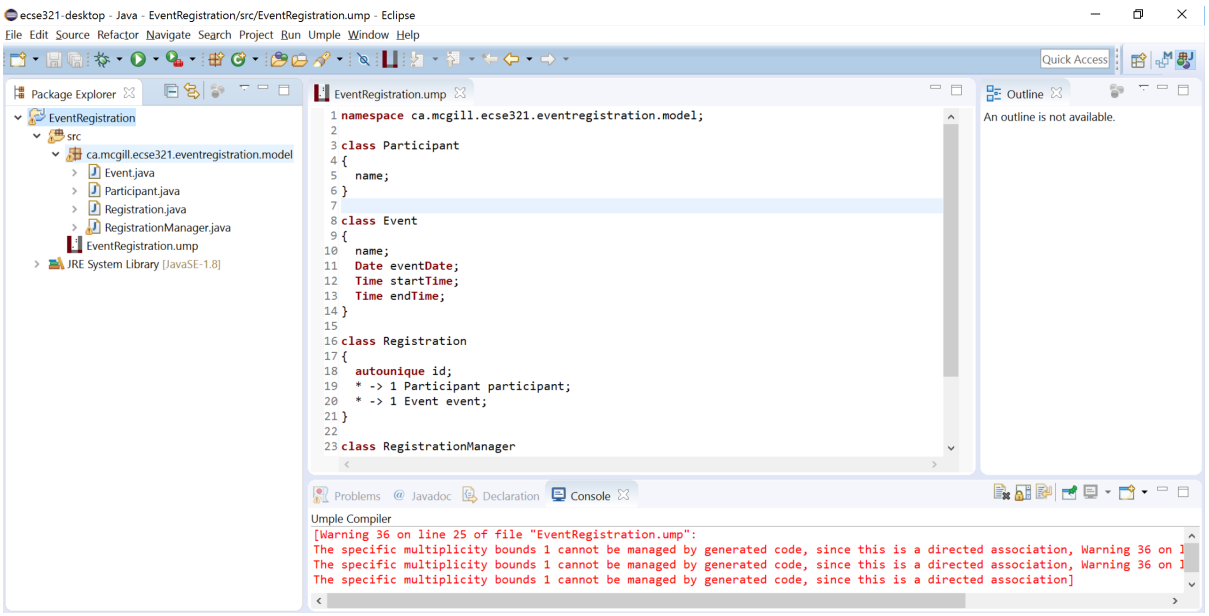


8. Refresh your project if the generated source code is not visible (e.g. **F5** or right click and **Refresh**).

   You may wish to set automatic refresh at **Window** | **Preferences** by checking *Refresh using native hooks* and *Refresh on access* in the **General** | **Workspace** preferences.



9. A new Java package `ca.mcgill.ecse321.eventregistration.model` containing four Java files (one for each class) should appear in **Package Explorer**



1.3. Testing Persistence

We implement a first unit test for testing persistence. The test creates instances of classes in the domain model and saves them to and loads them from an XML file using XStream.

1.3.1. Preparations

1. Open the project used previously.
2. Add the following JARs (available from myCourses) to a new `lib` folder and then add them to the build path:
   - `xstream-1.4.7.jar`
   - `xmlpull-1.1.3.1.jar`
   - `xpp3_min-1.1.4c.jar`
   - `jdatepicker-1.3.4.jar`
3. In addition, add the following class to your `src` folder within the `ca.mcgill.ecse321.eventregistration.persistence` package. This class is provided because you can easily find the code for writing to and reading from an XML file using the *XStream* library.

```
package ca.mcgill.ecse321.eventregistration.persistence;

import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

import com.thoughtworks.xstream.XStream;

import ca.mcgill.ecse321.eventregistration.model.Event;
import ca.mcgill.ecse321.eventregistration.model.Participant;
import ca.mcgill.ecse321.eventregistration.model.Registration;
import ca.mcgill.ecse321.eventregistration.model.RegistrationManager;

public abstract class PersistenceXStream {

    private static XStream xstream = new XStream();
    private static String filename = "data.xml";

    public static RegistrationManager initializeModelManager(String fileName) {
        // Initialization for persistence
        RegistrationManager rm;
        setFilename(fileName);
        setAlias("event", Event.class);
        setAlias("participant", Participant.class);
        setAlias("registration", Registration.class);
        setAlias("manager", RegistrationManager.class);

        // load model if exists, create otherwise
        File file = new File(fileName);
```

```
        if (file.exists()) {
            rm = (RegistrationManager) loadFromXMLwithXStream();
        } else {
            try {
                file.createNewFile();
            } catch (IOException e) {
                e.printStackTrace();
                System.exit(1);
            }
            rm = new RegistrationManager();
            saveToXMLwithXStream(rm);
        }
        return rm;

    }

    public static boolean saveToXMLwithXStream(Object obj) {
        xstream.setMode(XStream.ID_REFERENCES);
        String xml = xstream.toXML(obj); // save our xml file

        try {
            FileWriter writer = new FileWriter(filename);
            writer.write(xml);
            writer.close();
            return true;
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
    }

    public static Object loadFromXMLwithXStream() {
        xstream.setMode(XStream.ID_REFERENCES);
        try {
            FileReader fileReader = new FileReader(filename); // load our xml file
            return xstream.fromXML(fileReader);
        }
        catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }

    public static void setAlias(String xmlTagName, Class<?> className) {
        xstream.alias(xmlTagName, className);
    }

    public static void setFilename(String fn) {
        filename = fn;
    }
```

```
}
```

1.3.2. Towards unit testing of persistence

We develop an (integration) test which creates instances of classes in the domain model and saves them to an XML file and loads them from an XML file using XStream.

1. Add the JUnit 4 library to your build path.
   - If you enter `@Test` as an annotation, Eclipse will take care of that automatically.
2. Create a new source folder for the test cases by **Right click** on the project and **New | Source folder**. Name it to **test**.
   - An existing folder can be used as a source folder by **Right click** on the folder and **Build path | Use as Source folder**
3. Create a Java package in the `test` folder by **Right click** on the folder and **New | Package**. Name it to `ca.mcgill.ecse321.eventregistration.persistence`.
4. Create a new JUnit test class by **Right click** on the package and **New | JUnit Test Case**, and name it to **TestPersistence**. Select the **setUp()** and **tearDown()** options. The wizard creates a skeleton with three methods:
   - **setUp():** is automatically executed before all tests, so it could be used to create some example event.
   - **tearDown():** is automatically executed after the tests, so it could be used to delete the events.
   - **test:** is an example test case.

     ```
     @Before public void setUp() throws Exception {  }
     @After public void tearDown() throws Exception { }
     @Test public void test() {...}
     ```

     - Alternatively, a normal Java class also can be used, but the *JUnit Test Case* wizard create the example skeleton with the correct annotations.
5. Create a registration manager *rm* in the test class, initialize it and add some test data to the manager in the **setUp()** method:

```
private RegistrationManager rm;

@Before
public void setUp() throws Exception {
    rm = new RegistrationManager();

    // create participants
    Participant pa = new Participant("Martin");
    Participant pa2 = new Participant("Jennifer");

    // create event
    Calendar c = Calendar.getInstance();
    c.set(2015,Calendar.SEPTEMBER,15,8,30,0);
    Date eventDate = new Date(c.getTimeInMillis());
    Time startTime = new Time(c.getTimeInMillis());
    c.set(2015,Calendar.SEPTEMBER,15,10,0,0);
    Time endTime = new Time(c.getTimeInMillis());
    Event ev = new Event("Concert", eventDate, startTime, endTime);

    // register participants to event
    Registration re = new Registration(pa, ev);
    Registration re2 = new Registration(pa2, ev);

    // manage registrations
    rm.addRegistration(re);
    rm.addRegistration(re2);
    rm.addEvent(ev);
    rm.addParticipant(pa);
    rm.addParticipant(pa2);
}
```

6. Add the following imports required for the **setUp()** method:
    - Eclipse should add them automatically if you click on the red quick fix icon.

```
import java.sql.Date;
import java.sql.Time;
import java.util.Calendar;

import ca.mcgill.ecse321.eventregistration.model.Event;
import ca.mcgill.ecse321.eventregistration.model.Participant;
import ca.mcgill.ecse321.eventregistration.model.Registration;
import ca.mcgill.ecse321.eventregistration.model.RegistrationManager;
```

7. In order to remove temporary data created during the test, we implement the **tearDown()** method as follows:

```
@After
public void tearDown() throws Exception {
    rm.delete();
}
```

1.3.3. Implement the test case

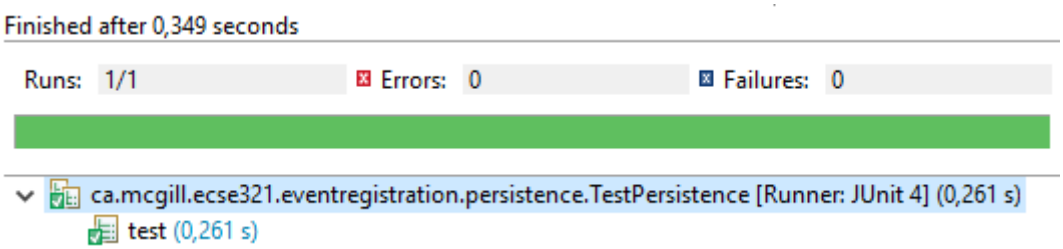
1. Implement the test case in the following way:

```
@Test
public void test() {
    // save model

PersistenceXStream.setFilename("test"+File.separator+"ca"+File.separator+"mcgill"+F
ile.separator+"ecse321"+File.separator+"eventregistration"+

File.separator+"persistence"+File.separator+"data.xml");
    PersistenceXStream.setAlias("event", Event.class);
    PersistenceXStream.setAlias("participant", Participant.class);
    PersistenceXStream.setAlias("registration", Registration.class);
    PersistenceXStream.setAlias("manager", RegistrationManager.class);
    if (!PersistenceXStream.saveToXMLwithXStream(rm))
        fail("Could not save file.");

    // clear the model in memory
    rm.delete();
    assertEquals(0, rm.getParticipants().size());
    assertEquals(0, rm.getEvents().size());
    assertEquals(0, rm.getRegistrations().size());

    // load model
    rm = (RegistrationManager) PersistenceXStream.loadFromXMLwithXStream();
    if (rm == null)
        fail("Could not load file.");

    // check participants
    assertEquals(2, rm.getParticipants().size());
    assertEquals("Martin", rm.getParticipant(0).getName());
    assertEquals("Jennifer", rm.getParticipant(1).getName());
    // check event
    assertEquals(1, rm.getEvents().size());
    assertEquals("Concert", rm.getEvent(0).getName());
    Calendar c = Calendar.getInstance();
    c.set(2015,Calendar.SEPTEMBER,15,8,30,0);
    Date eventDate = new Date(c.getTimeInMillis());
    Time startTime = new Time(c.getTimeInMillis());
    c.set(2015,Calendar.SEPTEMBER,15,10,0,0);
    Time endTime = new Time(c.getTimeInMillis());
    assertEquals(eventDate.toString(), rm.getEvent(0).getEventDate().toString());
    assertEquals(startTime.toString(), rm.getEvent(0).getStartTime().toString());
    assertEquals(endTime.toString(), rm.getEvent(0).getEndTime().toString());
    // check registrations
    assertEquals(2, rm.getRegistrations().size());
    assertEquals(rm.getEvent(0), rm.getRegistration(0).getEvent());
    assertEquals(rm.getParticipant(0), rm.getRegistration(0).getParticipant());
    assertEquals(rm.getEvent(0), rm.getRegistration(1).getEvent());
    assertEquals(rm.getParticipant(1), rm.getRegistration(1).getParticipant());
}
```

2. Add the following import for `File`:

```
import java.io.File;
```

3. To run the tests, press **Right click** on the file **Run as** | **Junit File**. A new window will appear, that shows the status of the test cases:



4. Additionally, after refreshing the project by **Right click** and **Refresh**, a **data.xml** file becomes visible in the `output` folder, which contains the persisted events:

```
<manager id="1">
  <registrations id="2">
    <registration id="3">
      <id>1</id>
      <participant id="4">
        <name>Martin</name>
      </participant>
      <event id="5">
        <name>Concert</name>
        <eventDate id="6">2015-09-15</eventDate>
        <startTime id="7">08:30:00</startTime>
        <endTime id="8">10:00:00</endTime>
      </event>
    </registration>
    <registration id="9">
      <id>2</id>
      <participant id="10">
        <name>Jennifer</name>
      </participant>
      <event reference="5"/>
    </registration>
  </registrations>
  <participants id="11">
    <participant reference="4"/>
    <participant reference="10"/>
  </participants>
  <events id="12">
    <event reference="5"/>
  </events>
</manager>
```

1.4. Controller - Part 1

1.4.1. Test-Driven Development of the Controller to Create Participant

We illustrate how to use the Test-Driven Development paradigm to implement one test for one method of the controller of the Event Registration Desktop Application and then implement this method of the controller itself.

1. In the `test` directory, create a new package `ca.mcgill.ecse321.eventregistration.controller`.
2. Right click the package and choose **New** | **JUnit Test Case**.
3. Name the test case `TestEventRegistrationController`. For the **Which method stubs would you like to create?** question, tick all enables options (**setUpBeforeClass()**, **setUp()**, **tearDownAfterClass()**, **tearDown()**) and click **Finish**.
4. Rename the `test()` method to `testCreateParticipant()`.
5. In the `src` directory and create a package `ca.mcgill.ecse321.eventregistration.controller`.
6. Create a new `EventRegistrationController` class in this package.
7. Add the constructor and the skeleton for the `createParticipant()` method.

   ```
   private RegistrationManager rm;

   public EventRegistrationController(RegistrationManager rm)
   {
     this.rm = rm;
   }

   public void createParticipant(String name)
   {
   }
   ```

   Save the file.

8. Go to the test class (`TestEventRegistrationController`) and add a basic test for the `testCreateParticipant()` method:

```
@Test
public void testCreateParticipant() {
  RegistrationManager rm = new RegistrationManager();
  assertEquals(0, rm.getParticipants().size()); // import Assert from the
`org.junit` package

  String name = "Oscar";

  EventRegistrationController erc = new EventRegistrationController(rm);
  erc.createParticipant(name);

  // check model in memory
  assertEquals(1, rm.getParticipants().size());
  assertEquals(name, rm.getParticipant(0).getName());
  assertEquals(0, rm.getEvents().size());
  assertEquals(0, rm.getRegistrations().size());

  RegistrationManager rm2 = (RegistrationManager)
PersistenceXStream.loadFromXMLwithXStream();

  // check file contents
  assertEquals(1, rm.getParticipants().size());
  assertEquals(name, rm.getParticipant(0).getName());
  assertEquals(0, rm.getEvents().size());
  assertEquals(0, rm.getRegistrations().size());
}
```

9. Implement the `setUpBeforeClass()` method:

```
PersistenceXStream.setFilename(
      "test" + File.separator + "ca" + File.separator + "mcgill" + File.separator +
"ecse321" + File.separator
              + "eventregistration" + File.separator + "controller" +
File.separator + "data.xml");
PersistenceXStream.setAlias("event", Event.class);
PersistenceXStream.setAlias("participant", Participant.class);
PersistenceXStream.setAlias("registration", Registration.class);
PersistenceXStream.setAlias("manager", RegistrationManager.class);
```

   Add the required imports and save the file.

10. Run the tests by right clicking the `TestEventRegistrationController` class and choosing **Run As** | **JUnit Test**. The test will fail and throw an error:

```
java.lang.AssertionError: expected:<1> but was:①
```

   This is expected, as the `EventRegistrationController` class is currently a dummy and its

`createParticipant()` method does not perform any operations.

11. Go to the `EventRegistrationController` class and implement the `createParticipant()` method:

```
Participant p = new Participant(name);
RegistrationManager rm = new RegistrationManager();
rm.addParticipant(p);
PersistenceXStream.saveToXMLwithXStream(rm);
```

    Save the file.

12. Go to the **JUnit** view and click **Rerun Test**. The test should run without errors.
13. Go to the `TestEventRegistrationController` class and select the last lines of the `testCreateParticipant()` method:

```
assertEquals(1, rm.getParticipants().size());
assertEquals(name, rm.getParticipant(0).getName());
assertEquals(0, rm.getEvents().size());
assertEquals(0, rm.getRegistrations().size());
```

    Right click and choose **Refactor** | **Extract Method...**. Set the method name to `checkResultParticipant`. Set the order of the parameters so that the first parameter is `name` and the second one is `rm2`.

    Note that the **Replace 1 additional occurrence of statements with method** checkbox is ticked and click **OK**.

14. As expected, the refactoring replaced both occurrences of the selected code snippet.
15. Rerun the tests to see if they successfully execute.

1.4.2. Create a Test Suite for All Tests

1. Right click the `test` directory and choose **New** | **Other** | **JUnit Test Suite**. For the **Package** field, click **Browse...** and choose the `ca.mcgill.ecse321.eventregistration` package. Leave the name as it is (`AllTests`) and click **Finish**.
2. For the `@SuiteClasses` annotation, add the following classes:

```
@SuiteClasses({ TestEventRegistrationController.class, TestPersistence.class })
```

3. In the **Package Explorer**, right click the `AllTests` class and choose **Run As** | **JUnit Test**.

1.5. Validation of Controller Input

We continue using the Test-Driven Development paradigm to implement further tests for the same `createParticipant()` method which now also tests invalid inputs into account and then add input validation to this method of the controller.

1.5.1. Introduce Input Exception

1. Create a new Java class `InvalidInputException` to be located within the `ca.mcgill.ecse321.eventregistration.controller` package and fill in as follows.

```
package ca.mcgill.ecse321.eventregistration.controller;

public class InvalidInputException extends Exception {
    private static final long serialVersionUID = -5633915762703837868L;
    public InvalidInputException(String errorMessage) {
        super(errorMessage);
    }
}
```

2. Add a `throws` declaration to method `createParticipant()` in class `EventRegistrationController`

```
public void createParticipant(String name) throws InvalidInputException
{
  Participant p = new Participant(name);
  rm.addParticipant(p);
  PersistenceXStream.saveToXMLwithXStream(rm);
}
```

1.5.2. Fix the existing test case

1. Open `TestEventRegistrationController.java` located in the `test` folder.
2. Surround the call to `createParticipant()` with a *try-catch* block. Click on the red error marker to do this step automatically. The final code should look as follows:

```
@Test
public void testCreateParticipant() {
  assertEquals(0, rm.getParticipants().size());

  String name = "Oscar";
  EventRegistrationController erc = new EventRegistrationController(rm);
  try {
    erc.createParticipant(name);
  } catch (InvalidInputException e) {
    // Check that no error occured
    fail();
  }

  RegistrationManager rm1 = rm;
  checkResultParticipant(name, rm1);

  RegistrationManager rm2 = (RegistrationManager)
PersistenceXStream.loadFromXMLwithXStream();
  checkResultParticipant(name, rm2);

  rm2.delete();
}
```

3. Execute the test case to check if every runs well. Right click on `TestEventRegistrationController` and **Run As.. | JUnit test**

1.5.3. Test method for empty participant

1. Create test method `testCreateParticipantNull()` in `TestEventRegistrationController.java` to check that the name of a participant cannot be null.

```
@Test
public void testCreateParticipantNull() {
  assertEquals(0, rm.getParticipants().size());
  String name = null;
  String error = null;

  EventRegistrationController erc = new EventRegistrationController(rm);
  try {
    erc.createParticipant(name);
  } catch (InvalidInputException e) {
    error = e.getMessage();
  }

  // check error
  assertEquals("Participant name cannot be empty!", error);

  // check no change in memory
  assertEquals(0, rm.getParticipants().size());
}
```

2. Re-run test cases. One of them should fail now.
3. Change method `createParticipant()` in `EventRegistrationController.java` to check that *name* is not **null**

```
public void createParticipant(String name) throws InvalidInputException
{
  if (name == null) {
  throw new InvalidInputException("Participant name cannot be empty!");
  }
  Participant p = new Participant(name);
  rm.addParticipant(p);
  PersistenceXStream.saveToXMLwithXStream(rm);
}
```

4. Re-run test cases, which should pass now.

1.5.4. Test methods for empty spaces

1. Create test method `testCreateParticipantEmpty()` to check that the name of the participant cannot be the empty string.

```
@Test
public void testCreateParticipantEmpty() {
  assertEquals(0, rm.getParticipants().size());

  String name = "";

  String error = null;
  EventRegistrationController erc = new EventRegistrationController(rm);
  try {
    erc.createParticipant(name);
  } catch (InvalidInputException e) {
    error = e.getMessage();
  }

  // check error
  assertEquals("Participant name cannot be empty!", error);

  // check no change in memory
  assertEquals(0, rm.getParticipants().size());
}
```

2. Create test method `testCreateParticipantSpaces()` to validate that the name of a participant cannot contain only spaces

```
@Test
public void testCreateParticipantSpaces() {
    assertEquals(0, rm.getParticipants().size());

    String name = " ";

    String error = null;
    EventRegistrationController erc = new EventRegistrationController(rm);
    try {
        erc.createParticipant(name);
    } catch (InvalidInputException e) {
        error = e.getMessage();
    }

    // check error
    assertEquals("Participant name cannot be empty!", error);

    // check no change in memory
    assertEquals(0, rm.getParticipants().size());
}
```

3. Re-run the test cases. Two of them should fail now.
4. Change method `createParticipant()` in `EventRegistrationController.java` to check that *name* is cannot contain only spaces. The final code should look like that.

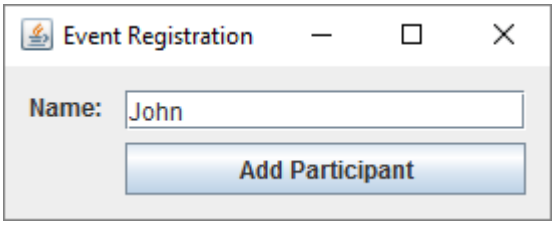
```
public void createParticipant(String name) throws InvalidInputException
{
  if (name == null || name.trim().length() == 0) {
    throw new InvalidInputException("Participant name cannot be empty!");
  }
  Participant p = new Participant(name);
  rm.addParticipant(p);
  PersistenceXStream.saveToXMLwithXStream(rm);
}
```

5. Re-run the test cases. All of them should pass now.

1.6. Creating the First View

1.6.1. View implementation

The goal of this section is to create the following a simple graphical user interface with the help of the Java Swing Framework for registering participants:



1. Create a new package to the **src** folder by **Right click** and **New** | **package**, and name it `ca.mcgill.ecse321.eventregistration.view`.
2. Create a new java class by **Right click** on the new package and **New** | **Java Class**, name it `EventRegistrationPage`.
3. Inherit the new class from `JFrame`:

```
package ca.mcgill.ecse321.eventregistration.view;

import javax.swing.JFrame;

public class EventRegistrationPage2 extends JFrame {

}
```

4. A warning appears on the class: *"The serializable class EventRegistrationPage2 does not declare a static final serialVersionUID field of type long"*. In the context menu, select **Add Generated serial version ID**.

```
private static final long serialVersionUID = -3813819647258555349L;
```

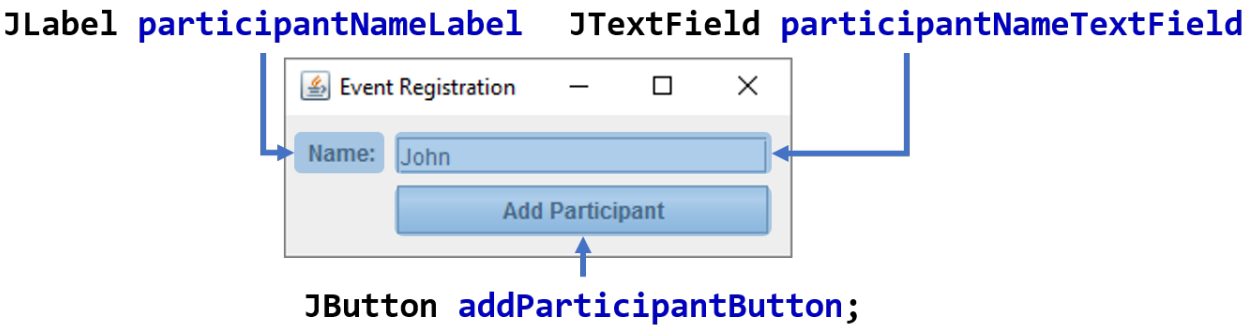
5. Add three private attributes to the class:

```
private JTextField participantNameTextField;
private JLabel participantNameLabel;
private JButton addParticipantButton;
```

6. Add the following imports:

```
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.JButton;
```

The attributes represent the following graphical elements:



7. Create a constructor with a parameter

```
private RegistrationManager rm;

/** Creates new form EventRegistrationPage */
public ParticipantRegistration(RegistrationManager rm) {
    this.rm = rm;
}
```

8. Create an initializer method that creates configures and layout the graphical elements:

```
private void initComponents() {
    // elements for participant
    participantNameTextField = new JTextField();
    participantNameLabel = new JLabel();
    addParticipantButton = new JButton();

    // global settings and listeners
    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    setTitle("Event Registration");

    participantNameLabel.setText("Name:");
    addParticipantButton.setText("Add Participant");

    // layout
    GroupLayout layout = new GroupLayout(getContentPane());
    getContentPane().setLayout(layout);
    layout.setAutoCreateGaps(true);
    layout.setAutoCreateContainerGaps(true);

    layout.setHorizontalGroup(
        layout.createSequentialGroup()
        .addComponent(participantNameLabel)
        .addGroup(layout.createParallelGroup()
            .addComponent(participantNameTextField, 200, 200, 400)
            .addComponent(addParticipantButton))
        );

    layout.linkSize(SwingConstants.HORIZONTAL, new java.awt.Component[]
{addParticipantButton, participantNameTextField});

    layout.setVerticalGroup(
        layout.createSequentialGroup()
        .addGroup(layout.createParallelGroup()
            .addComponent(participantNameLabel)
            .addComponent(participantNameTextField))
        .addComponent(addParticipantButton)
        );

    pack();
}
```

9. Add the following imports:

```
import javax.swing.GroupLayout;
import javax.swing.SwingConstants;
import javax.swing.WindowConstants;
```

10. Call the `initializer` method in the constructor:

```
public ParticipantRegistration(RegistrationManager rm) {
    this.rm = rm;
    initComponents();
}
```

11. Whenever something changes in the model, the view have to be refreshed. Currently it simply clears the text field. Create the following method:

```
private void refreshData() {
    participantNameTextField.setText("");
    pack();
}
```

12. We need to react to the press of the *Add Participant* button. First, lets create a method that reads the value of the text field, and execute the change in the model, and clear the text from the text field:

```
private void addParticipantButtonActionPerformed() {
// create and call the controller
EventRegistrationController erc = new EventRegistrationController(rm);
try {
    erc.createParticipant(participantNameTextField.getText());
} catch (InvalidInputException e) {
    // At that point, we ignore the exception
}
refreshData();
}
```

13. Register the previous method to the `addParticipantButton` button by adding the following line to the initializer:

```
addParticipantButton.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        addParticipantButtonActionPerformed();
    }
});
```

1.6.2. Running the application

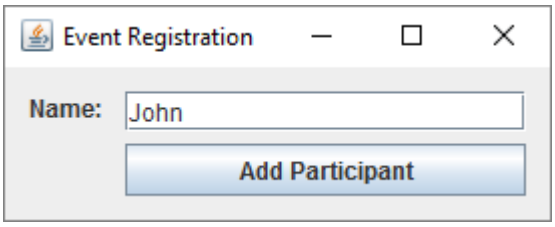
1. Create a new package to the **src** folder by **Right click** and **New** | **package**, and name it `ca.mcgill.ecse321.eventregistration.application`.
2. Create a new java class by **Right click** and **New** | **class**, name it `EventRegistration`, and select the `public static void main(String[] args)` option in the wizard.
    - This option creates a Java class with a `main` method.

3. Fill the new class as follows:

```
public class EventRegistration {
    private static String fileName = "output/eventregistration.xml";

    public static void main(String[] args) {
      // load model
          final RegistrationManager rm =
PersistenceXStream.initializeModelManager(fileName);

      // start UI
      java.awt.EventQueue.invokeLater(new Runnable() {
          public void run() {
              new EventRegistrationPage(rm).setVisible(true);
          }
      });
    }
}
```

   - This piece of code first initializes the `RegistrationManager` with an XML file in the `main` method
   - Then it starts the UI in the second step with a new **Runnable** anonym instance.

4. Run the application by **Right click** on `EventRegistration.java` **Run | Java application**. The following window will appear:



* By adding a new name, the application save it as a participant.

+

```
<manager id="1">
  <registrations id="2"/>
  <participants id="3">
    <participant id="4">
      <name>John</name>
    </participant>
  </participants>
  <events id="5"/>
</manager>
```

1.6.3. Error handling

1. Add two attribute to the class:

```
private String error = null;
private JLabel errorMessage;
```

2. Initialize the label that shows the error message in the `initComponents` method:

```
private void initComponents() {
    // elements for error message
    errorMessage = new JLabel();
    errorMessage.setForeground(Color.RED);
    ...
    // Note that the groups are changed
    layout.setHorizontalGroup(
        // error message is adde here
        layout.createParallelGroup()
        .addComponent(errorMessage)
        .addGroup(layout.createSequentialGroup()
        .addComponent(participantNameLabel)
        .addGroup(layout.createParallelGroup()
            .addComponent(participantNameTextField, 200, 200, 400)
            .addComponent(addParticipantButton))
        ));
    ...
    layout.setVerticalGroup(
        layout.createSequentialGroup()
        // error message is adde here
        .addComponent(errorMessage)
        .addGroup(layout.createParallelGroup()
            .addComponent(participantNameLabel)
            .addComponent(participantNameTextField))
        .addComponent(addParticipantButton)
        );
}
```

   - Note that The layout groups in the horizontal alignment has changed a lot.

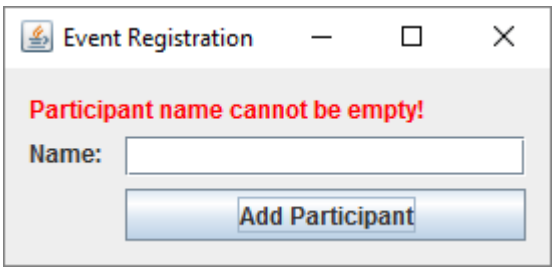
3. Upon refresh, the error message should be also cleaned:

```
private void refreshData() {
    // error
    errorMessage.setText(error);
    if (error == null || error.length() == 0) {
        // participant
        participantNameTextField.setText("");
    }
    // this is needed because the size of the window changes depending on whether
an error message is shown or not
    pack();
}
```

4. And finally, if there is any problem with the name of the participant, the error message is written in `error` string:

```
private void addParticipantButtonActionPerformed(java.awt.event.ActionEvent evt) {
    // call the controller
    EventRegistrationController erc = new EventRegistrationController(rm);
    error = null;
    try {s
        erc.createParticipant(participantNameTextField.getText());
    } catch (InvalidInputException e) {
        error = e.getMessage();
    }
    // update visuals
    refreshData();
}
```

5. When a participant name is malformed (e.g. empty), then the application will show an error message:



1.7. Controller Input Validation (Part 2)

We use the Test-Driven Development paradigm to implement the “everything is ok” scenarios for the remaining methods of the controller of the Event Registration Desktop Application.

1.7.1. Test-Driven Development of Create Event of the Controller

1. Go to the `EventRegistrationController` class and add the `createEvent()` method:

```
public void createEvent(String name, Date date, Time startTime, Time endTime) {
 // To be completed
}
```

   - Import the `Date` and the `Time` classes from the `java.sql` package.

2. Go to the `TestEventRegistrationController` class and add the `testCreateEvent()` method:

```
public void testCreateEvent() {
  RegistrationManager rm = new RegistrationManager();
  assertEquals(0, rm.getEvents().size());

  String name = "Soccer Game";
  Calendar c = Calendar.getInstance();
  c.set(2017, Calendar.MARCH, 16, 9, 0, 0);
  Date eventDate = new Date(c.getTimeInMillis());
  Time startTime = new Time(c.getTimeInMillis());
  c.set(2017, Calendar.MARCH, 16, 10, 30, 0);
  Time endTime = new Time(c.getTimeInMillis());

  EventRegistrationController erc = new EventRegistrationController(rm);
  try {
      erc.createEvent(name, eventDate, startTime, endTime);
  } catch (InvalidInputException e) {
      fail();
  }

  checkResultEvent(name, eventDate, startTime, endTime, rm);

  RegistrationManager rm2 = (RegistrationManager)
PersistenceXStream.loadFromXMLwithXStream();
  checkResultEvent(name, eventDate, startTime, endTime, rm2);
  rm2.delete();

}
```

3. The code will not yet compile, as we need to implement the `checkResultEvent()` method.

```
private void checkResultEvent(String name, Date eventDate, Time startTime, Time
endTime, RegistrationManager rm2)   {
  assertEquals(0, rm2.getParticipants().size());
  assertEquals(1, rm2.getEvents().size());
  assertEquals(name, rm2.getEvent(0).getName());
  assertEquals(eventDate.toString(), rm2.getEvent(0).getEventDate().toString());
  assertEquals(startTime.toString(), rm2.getEvent(0).getStartTime().toString());
  assertEquals(endTime.toString(), rm2.getEvent(0).getEndTime().toString());
  assertEquals(0, rm2.getRegistrations().size());
}
```

4. Run the JUnit tests again. The `testCreateEvent()` method should fail as it is not yet implemented.
5. Go to the `EventRegistrationController` class and implement the `createEvent()` method:

```
public void createEvent(String name, Date date, Time startTime, Time endTime)
throws InvalidInputException
{
    Event e = new Event(name, date, startTime, endTime);
    RegistrationManager rm = new RegistrationManager();
    rm.addEvent(e);
    PersistenceXStream.saveToXMLwithXStream(rm);
}
```

6. Rerun the JUnit tests. They should now complete without errors or failures.

1.7.2. Test-Driven Development of Register Participant

1. Create a `testRegister()` method:

```
@Test public void testRegister() {
  RegistrationManager rm = new RegistrationManager();
  assertEquals(0, rm.getRegistrations().size());

  String nameP = "Oscar";
  Participant participant = new Participant(nameP);
  rm.addParticipant(participant);
  assertEquals(1, rm.getParticipants().size());

  String nameE = "Soccer Game";
  Calendar c = Calendar.getInstance();
  c.set(2017, Calendar.MARCH, 16, 9, 0, 0);
  Date eventDate = new Date(c.getTimeInMillis());
  Time startTime = new Time(c.getTimeInMillis());
  c.set(2017, Calendar.MARCH, 16, 10, 30, 0);
  Time endTime = new Time(c.getTimeInMillis());
  Event event = new Event(nameE, eventDate, startTime, endTime);
  rm.addEvent(event);
  assertEquals(1, rm.getEvents().size());

  EventRegistrationController erc = new EventRegistrationController(rm);
  try {
      erc.register(participant, event);
  } catch (InvalidInputException e) {
      fail();
  }

  checkResultRegister(nameP, nameE, eventDate, startTime, endTime, rm);

  RegistrationManager rm2 = (RegistrationManager)
PersistenceXStream.loadFromXMLwithXStream();
  // check file contents
  checkResultRegister(nameP, nameE, eventDate, startTime, endTime, rm2);
  rm2.delete();
}
```

2. Implement the `checkResultRegister()` method:

```
private void checkResultRegister(String nameP, String nameE, Date eventDate, Time
startTime, Time endTime,
        RegistrationManager rm2)
{
    assertEquals(1, rm2.getParticipants().size());
    assertEquals(nameP, rm2.getParticipant(0).getName());
    assertEquals(1, rm2.getEvents().size());
    assertEquals(nameE, rm2.getEvent(0).getName());
    assertEquals(eventDate.toString(), rm2.getEvent(0).getEventDate().toString());
    assertEquals(startTime.toString(), rm2.getEvent(0).getStartTime().toString());
    assertEquals(endTime.toString(), rm2.getEvent(0).getEndTime().toString());
    assertEquals(1, rm2.getRegistrations().size());
    assertEquals(rm2.getEvent(0), rm2.getRegistration(0).getEvent());
    assertEquals(rm2.getParticipant(0), rm2.getRegistration(0).getParticipant());
}
```

3. Run the JUnit tests again. The `testRegister()` methods will fail as it is not yet implemented.
4. Go to the `EventRegistrationController` class and implement the `register()` method:

```
public void register(Participant participant, Event event) throws
InvalidInputException
{
    RegistrationManager rm = new RegistrationManager();
    Registration r = new Registration(participant, event);
    rm.addRegistration(r);
    PersistenceXStream.saveToXMLwithXStream(rm);
}
```

5. Rerun the JUnit tests. They should now complete without errors or failures.

1.8. Controller Input Validation (Part 2)

We use the Test-Driven Development paradigm to implement the scenarios with invalid input for the remaining methods of the controller of the Event Registration Desktop Application.

1.8.1. Test-Driven Development of Input Validation for Create Event

1. Implement the `testCreateEventNull()` method:

```
@Test
public void testCreateEventNull() { assertEquals(0, rm.getRegistrations().size());

  String name = null;
  Date eventDate = null;
  Time startTime = null;
  Time endTime = null;

  String error = null;
  EventRegistrationController erc = new EventRegistrationController(rm);
  try {
      erc.createEvent(name, eventDate, startTime, endTime);
  } catch (InvalidInputException e) {
      error = e.getMessage();
  }

  // check error
  assertEquals(
          "Event name cannot be empty! Event date cannot be empty! Event start time
cannot be empty! Event end time cannot be empty!",
          error);
  // check model in memory
  assertEquals(0, rm.getEvents().size());
}
```

2. Implement the `testCreateEventEmpty()` method:

```
@Test public void testCreateEventEmpty() {
  assertEquals(0, rm.getEvents().size());

  String name = "";
  Calendar c = Calendar.getInstance();
  c.set(2017, Calendar.FEBRUARY, 16, 10, 00, 0);
  Date eventDate = new Date(c.getTimeInMillis());
  Time startTime = new Time(c.getTimeInMillis());
  c.set(2017, Calendar.FEBRUARY, 16, 11, 30, 0);
  Time endTime = new Time(c.getTimeInMillis());

  String error = null;
  EventRegistrationController erc = new EventRegistrationController(rm);
  try {
      erc.createEvent(name, eventDate, startTime, endTime);
  } catch (InvalidInputException e) {
      error = e.getMessage();
  }

  // check error
  assertEquals("Event name cannot be empty!", error);
  // check model in memory
  assertEquals(0, rm.getEvents().size());
}
```

3. Create the `testCreateEventSpaces()` method:

```
@Test public void testCreateEventSpaces() {
  assertEquals(0, rm.getEvents().size());

  String name = " ";
  Calendar c = Calendar.getInstance();
  c.set(2016, Calendar.OCTOBER, 16, 9, 00, 0);
  Date eventDate = new Date(c.getTimeInMillis());
  Time startTime = new Time(c.getTimeInMillis());
  c.set(2016, Calendar.OCTOBER, 16, 10, 30, 0);
  Time endTime = new Time(c.getTimeInMillis());

  String error = null;
  EventRegistrationController erc = new EventRegistrationController(rm);
  try {
      erc.createEvent(name, eventDate, startTime, endTime);
  } catch (InvalidInputException e) {
      error = e.getMessage();
  }
  // check error
  assertEquals("Event name cannot be empty!", error);
  // check model in memory
  assertEquals(0, rm.getEvents().size());
}
```

4. Create the `testCreateEventEndTimeBeforeStartTime()` method.

```
@Test public void testCreateEventEndTimeBeforeStartTime() {
assertEquals(0, rm.getEvents().size());

  String name = "Soccer Game";
  Calendar c = Calendar.getInstance();
  c.set(2016, Calendar.OCTOBER, 16, 9, 00, 0);
  Date eventDate = new Date(c.getTimeInMillis());
  Time startTime = new Time(c.getTimeInMillis());
  c.set(2016, Calendar.OCTOBER, 16, 8, 59, 59);
  Time endTime = new Time(c.getTimeInMillis());

  String error = null;
  EventRegistrationController erc = new EventRegistrationController(rm);
  try {
      erc.createEvent(name, eventDate, startTime, endTime);
  } catch (InvalidInputException e) {
      error = e.getMessage();
  }

  // check error
  assertEquals("Event end time cannot be before event start time!", error);

  // check model in memory
  assertEquals(0, rm.getEvents().size());
}
```

5. Save the files and run the JUnit tests. The new tests will fail:
    - `testCreateEventNull()`
    - `testCreateEventSpaces()`
    - `testCreateEventEndTimeBeforeStartTime()`
    - `testCreateEventEmpty()`
6. To fix this, we need to implement the expected error messages. Add the following checks to the beginning of the `createEvent()` method:

```
String error = "";
if (name == null || name.trim().length() == 0)
  error = error + "Event name cannot be empty! ";
if (date == null)
  error = error + "Event date cannot be empty! ";
if (startTime == null)
  error = error + "Event start time cannot be empty! ";
if (endTime == null)
  error = error + "Event end time cannot be empty! ";
if (endTime != null && startTime != null && endTime.getTime() <
startTime.getTime())
  error = error + "Event end time cannot be before event start time!";
error = error.trim();
if (error.length() > 0)
  throw new InvalidInputException(error);
```

7. Rerun the tests, which should now run without any errors or failures.

1.8.2. Test-Driven Development of Input Validation for Register Participant

1. Add the `testRegisterNull()` method:

```
@Test public void testRegisterNull() {
  assertEquals(0, rm.getRegistrations().size());

  Participant participant = null;
  assertEquals(0, rm.getParticipants().size());

  Event event = null;
  assertEquals(0, rm.getEvents().size());

  String error = null;
  EventRegistrationController erc = new EventRegistrationController(rm);
  try {
      erc.register(participant, event);
  } catch (InvalidInputException e) {
      error = e.getMessage();
  }

  // check error
  assertEquals("Participant needs to be selected for registration! Event needs to
be selected for registration!",
          error);

  // check model in memory
  assertEquals(0, rm.getRegistrations().size());
  assertEquals(0, rm.getParticipants().size());
  assertEquals(0, rm.getEvents().size());
}
```

2. Add the `testRegisterParticipantAndEventDoNotExist()` method.

```
@Test
public void testRegisterParticipantAndEventDoNotExist() {
    assertEquals(0, rm.getRegistrations().size());

    String nameP = "Oscar";
    Participant participant = new Participant(nameP);
    assertEquals(0, rm.getParticipants().size());

    String nameE = "Soccer Game";
    Calendar c = Calendar.getInstance();
    c.set(2016, Calendar.OCTOBER, 16, 9, 00, 0);
    Date eventDate = new Date(c.getTimeInMillis());
    Time startTime = new Time(c.getTimeInMillis());
    c.set(2016, Calendar.OCTOBER, 16, 10, 30, 0);
    Time endTime = new Time(c.getTimeInMillis());
    Event event = new Event(nameE, eventDate, startTime, endTime);
    assertEquals(0, rm.getEvents().size());

    String error = null;
    EventRegistrationController erc = new EventRegistrationController(rm);
    try {
        erc.register(participant, event);
    } catch (InvalidInputException e) {
        error = e.getMessage();
    }

    // check error
    assertEquals("Participant does not exist! Event does not exist!", error);

    // check model in memory
    assertEquals(0, rm.getRegistrations().size());
    assertEquals(0, rm.getParticipants().size());
    assertEquals(0, rm.getEvents().size());
}
```

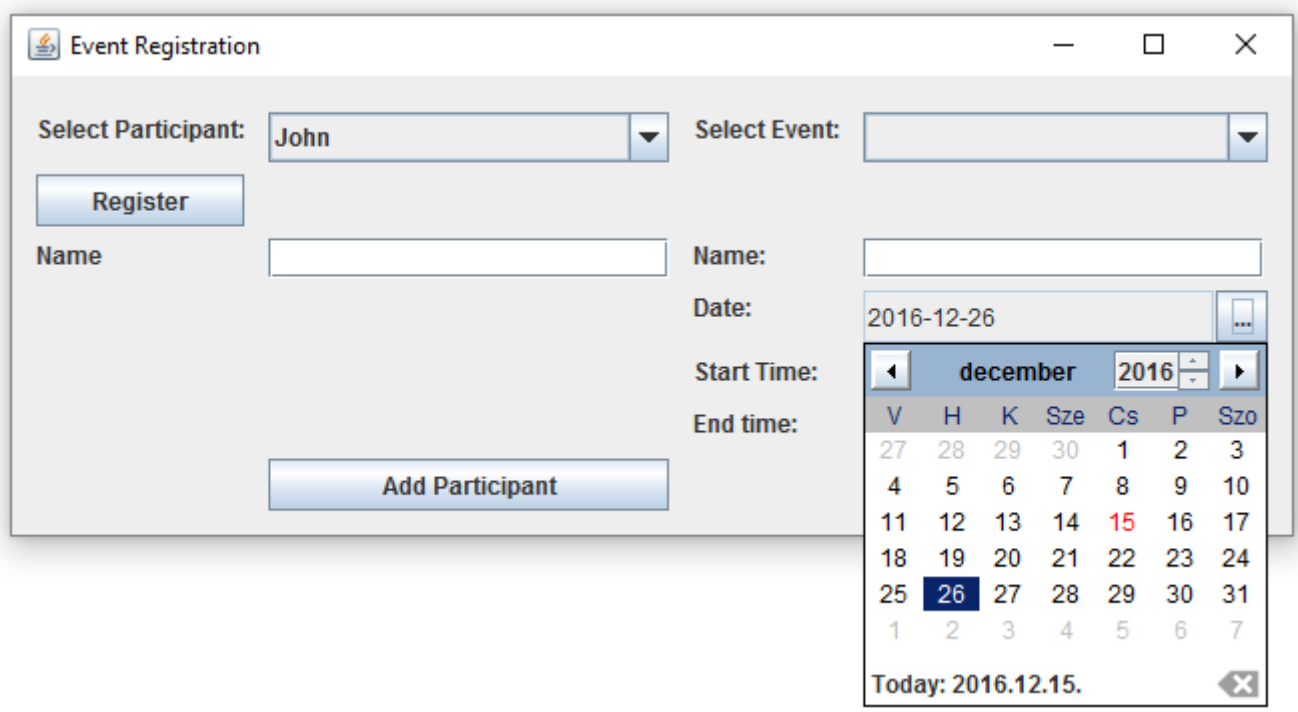
3. Save the files and run the JUnit tests. The new tests will fail:
    - `testRegisterNull()`
    - `testRegisterParticipantAndEventDoNotExist()`
4. To fix these, go to the `EventRegistrationController` class, and add the following checks to the beginning of the `register()` method:

```
String error = "";
if (participant == null)
  error = error + "Participant needs to be selected for registration! ";
else if (!rm.getParticipants().contains(participant))
  error = error + "Participant does not exist! ";
if (event == null)
  error = error + "Event needs to be selected for registration!";
else if (!rm.getEvents().contains(event))
  error = error + "Event does not exist!";
error = error.trim();
if (error.length() > 0)
  throw new InvalidInputException(error);

Registration r = new Registration(participant, event);
rm.addRegistration(r);
PersistenceXStream.saveToXMLwithXStream(rm);
```

5. Rerun the tests, which should now work without any errors or failures.

1.9. Creating the View (Part 2)

We implement the User Interface of the remaining functionality of the Event Registration Desktop Application with the help of the Java Swing Framework. A screenshot of the complete application is illustrated below:



1.9.1. A helper class

1. Create a new class in the `ca.mcgill.ecse321.eventregistration.view` package.

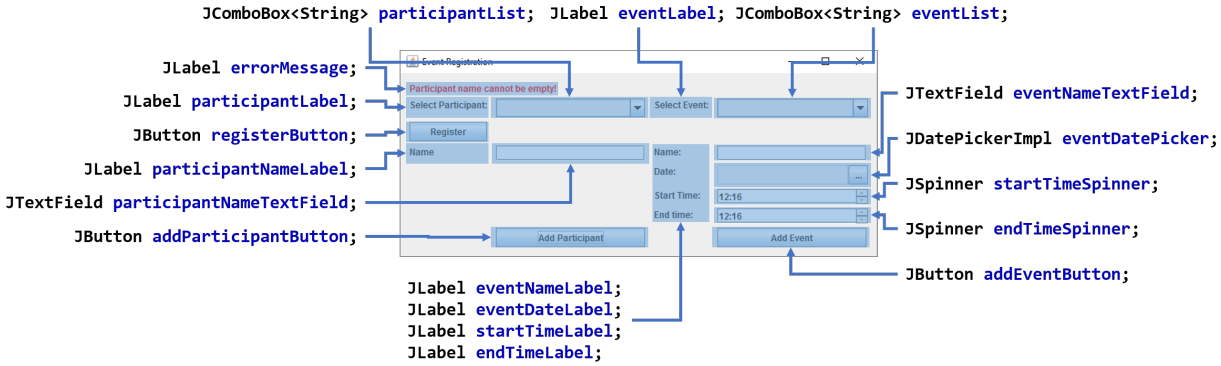
```
package ca.mcgill.ecse321.eventregistration.view;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import javax.swing.JFormattedTextField.AbstractFormatter;

public class DateLabelFormatter extends AbstractFormatter {
  private static final long serialVersionUID = -2169252224419341678L;

  private String datePattern = "yyyy-MM-dd";
  private SimpleDateFormat dateFormatter = new SimpleDateFormat(datePattern);

  @Override
  public Object stringToValue(String text) throws ParseException {
      return dateFormatter.parseObject(text);
  }

  @Override
  public String valueToString(Object value) throws ParseException {
      if (value != null) {
          Calendar cal = (Calendar) value;
          return dateFormatter.format(cal.getTime());
      }

      return "";
  }

}
```

1.9.2. The UI elements

1. Create a class called `EventRegistrationPage` by right clicking on the package `ca.mcgill.ecse321.eventregistration.view`, **New** | **Class**.
2. Add the following elements to the new class:

```
// UI elements
private JLabel errorMessage;
private JComboBox<String> participantList;
private JLabel participantLabel;
private JComboBox<String> eventList;
private JLabel eventLabel;
private JButton registerButton;
private JTextField participantNameTextField;
private JLabel participantNameLabel;
private JButton addParticipantButton;
private JTextField eventNameTextField;
private JLabel eventNameLabel;
private JDatePickerImpl eventDatePicker;
private JLabel eventDateLabel;
private JSpinner startTimeSpinner;
private JLabel startTimeLabel;
private JSpinner endTimeSpinner;
private JLabel endTimeLabel;
private JButton addEventButton;
```

- The attributes represent the following graphical elements:



3. Add the following imports:

```
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JSpinner;
import javax.swing.JTextField;

import org.jdatepicker.impl.JDatePickerImpl;
```

4. Additionally, the application uses the following attributes:

```
private RegistrationManager rm;

// data elements
private String error = null;
private Integer selectedParticipant = -1;
private Integer selectedEvent = -1;
```

5. Create a method that initialize and layout the graphical elements:

```
private void initComponents() {
    // elements for error message
    errorMessage = new JLabel();
    errorMessage.setForeground(Color.RED);

    // elements for registration
    participantList = new JComboBox<String>(new String[0]);

    participantLabel = new JLabel();
    eventList = new JComboBox<String>(new String[0]);

    eventLabel = new JLabel();
    registerButton = new JButton();

    // elements for participant
    participantNameTextField = new JTextField();
    participantNameLabel = new JLabel();
    addParticipantButton = new JButton();

    // elements for event
    eventNameTextField = new JTextField();
    eventNameLabel = new JLabel();

    SqlDateModel model = new SqlDateModel();
    Properties p = new Properties();
    p.put("text.today", "Today");
    p.put("text.month", "Month");
    p.put("text.year", "Year");
    JDatePanelImpl datePanel = new JDatePanelImpl(model, p);
    eventDatePicker = new JDatePickerImpl(datePanel, new DateLabelFormatter());

    eventDateLabel = new JLabel();
    startTimeSpinner = new JSpinner( new SpinnerDateModel() );
    JSpinner.DateEditor startTimeEditor = new JSpinner.DateEditor(startTimeSpinner,
"HH:mm");
    startTimeSpinner.setEditor(startTimeEditor); // will only show the current time
    startTimeLabel = new JLabel();
    endTimeSpinner = new JSpinner( new SpinnerDateModel() );
    JSpinner.DateEditor endTimeEditor = new JSpinner.DateEditor(endTimeSpinner,
"HH:mm");
```

```
    endTimeSpinner.setEditor(endTimeEditor); // will only show the current time
    endTimeLabel = new JLabel();
    addEventButton = new JButton();

    // global settings and listeners
    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    setTitle("Event Registration");

    participantLabel.setText("Select Participant:");
    eventLabel.setText("Select Event:");
    registerButton.setText("Register");

    participantNameLabel.setText("Name");
    addParticipantButton.setText("Add Participant");

    eventNameLabel.setText("Name:");
    eventDateLabel.setText("Date:");
    startTimeLabel.setText("Start Time:");
    endTimeLabel.setText("End time:");
    addEventButton.setText("Add Event");

    // layout
    GroupLayout layout = new GroupLayout(getContentPane());
    getContentPane().setLayout(layout);
    layout.setAutoCreateGaps(true);
    layout.setAutoCreateContainerGaps(true);
    layout.setHorizontalGroup(

            layout.createParallelGroup()
            .addComponent(errorMessage)
            .addGroup(layout.createSequentialGroup()
                    .addGroup(layout.createParallelGroup()
                            .addComponent(participantLabel)
                            .addComponent(registerButton)
                            .addComponent(participantNameLabel))
                    .addGroup(layout.createParallelGroup()
                            .addComponent(participantList)
                            .addComponent(participantNameTextField, 200, 200, 400)
                            .addComponent(addParticipantButton))
                    .addGroup(layout.createParallelGroup()
                            .addComponent(eventLabel)
                            .addComponent(eventNameLabel)
                            .addComponent(eventDateLabel)
                            .addComponent(startTimeLabel)
                            .addComponent(endTimeLabel))
                    .addGroup(layout.createParallelGroup()
                            .addComponent(eventList)
                            .addComponent(eventNameTextField, 200, 200, 400)
                            .addComponent(eventDatePicker)
                            .addComponent(startTimeSpinner)
                            .addComponent(endTimeSpinner)
```

```
                            .addComponent(addEventButton)))
        );

    layout.linkSize(SwingConstants.HORIZONTAL, new java.awt.Component[]
{registerButton, participantLabel});
    layout.linkSize(SwingConstants.HORIZONTAL, new java.awt.Component[]
{addParticipantButton, participantNameTextField});
    layout.linkSize(SwingConstants.HORIZONTAL, new java.awt.Component[]
{addEventButton, eventNameTextField});

    layout.setVerticalGroup(
            layout.createSequentialGroup()
            .addComponent(errorMessage)
            .addGroup(layout.createParallelGroup()
                    .addComponent(participantLabel)
                    .addComponent(participantList)
                    .addComponent(eventLabel)
                    .addComponent(eventList))
            .addComponent(registerButton)
            .addGroup(layout.createParallelGroup()
                    .addComponent(participantNameLabel)
                    .addComponent(participantNameTextField)
                    .addComponent(eventNameLabel)
                    .addComponent(eventNameTextField))
            .addGroup(layout.createParallelGroup()
                    .addComponent(eventDateLabel)
                    .addComponent(eventDatePicker))
            .addGroup(layout.createParallelGroup()
                    .addComponent(startTimeLabel)
                    .addComponent(startTimeSpinner))
            .addGroup(layout.createParallelGroup()
                    .addComponent(endTimeLabel)
                    .addComponent(endTimeSpinner))
            .addGroup(layout.createParallelGroup()
                    .addComponent(addParticipantButton)
                    .addComponent(addEventButton))
        );

    pack();
}
```

6. Upon a change in the data model, the user interface have to be refreshed. Create the following method that loads the model and fill the UI elements with the correct content:

```
private void refreshData() {
    // error
    errorMessage.setText(error);
    if (error == null || error.length() == 0) {
      // participant list
            participantList.removeAllItems();
            for (Participant p : rm.getParticipants()) {
                participantList.addItem(p.getName());
            }
            selectedParticipant = -1;
            participantList.setSelectedIndex(selectedParticipant);
            // event list
            eventList.removeAllItems();
            for (Event e : rm.getEvents()) {
                eventList.addItem(e.getName());
            }
            selectedEvent = -1;
            eventList.setSelectedIndex(selectedEvent);
            // participant
            participantNameTextField.setText("");
            // event
            eventNameTextField.setText("");
            eventDatePicker.getModel().setValue(null);
            startTimeSpinner.setValue(new Date());
            endTimeSpinner.setValue(new Date());
    }

    // this is needed because the size of the window changes depending on whether
an error message is shown or not
    pack();
}
```

7. Create a constructor that (1) sets the registration manager, (2) initialize the graphical components and (3) fills the user interface with the persons and events available in the model:

```
public EventRegistrationPage(RegistrationManager aRegMan) {
    rm = aRegMan;
    initComponents();
    refreshData();
}
```

1.9.3. User actions

1. Now create methods for the user actions:

```
private void addParticipantButtonActionPerformed() {
    // call the controller
```

```
    EventRegistrationController erc = new EventRegistrationController(rm);
    error = null;
    try {
        erc.createParticipant(participantNameTextField.getText());
    } catch (InvalidInputException e) {
        error = e.getMessage();
    }
    // update visuals
    refreshData();
}

private void addEventButtonActionPerformed() {
    // call the controller
    EventRegistrationController erc = new EventRegistrationController(rm);
    // JSpinner actually returns a date and time
    // force the same date for start and end time to ensure that only the times
differ
    Calendar calendar = Calendar.getInstance();
    calendar.setTime((Date) startTimeSpinner.getValue());
    calendar.set(2000, 1, 1);
    Time startTime = new Time(calendar.getTime().getTime());
    calendar.setTime((Date) endTimeSpinner.getValue());
    calendar.set(2000, 1, 1);
    Time endTime = new Time(calendar.getTime().getTime());
    error = null;
    try {
        erc.createEvent(eventNameTextField.getText(), (java.sql.Date)
eventDatePicker.getModel().getValue(), startTime, endTime);
    } catch (InvalidInputException e) {
        error = e.getMessage();
    }
    // update visuals
    refreshData();
}

private void registerButtonActionPerformed() {
    error = "";
    if (selectedParticipant < 0)
        error = error + "Participant needs to be selected for registration! ";
    if (selectedEvent < 0)
        error = error + "Event needs to be selected for registration!";
    error = error.trim();
    if (error.length() == 0) {
        // call the controller
        EventRegistrationController erc = new EventRegistrationController(rm);
        try {
            erc.register(rm.getParticipant(selectedParticipant),
rm.getEvent(selectedEvent));
        } catch (InvalidInputException e) {
            error = e.getMessage();
        }
```

```
    }
    // update visuals
    refreshData();
}
```

2. Register the methods as action handlers in the initialization method:

```
private void initComponents() {
    ...
    participantList.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            JComboBox<String> cb = (JComboBox<String>) evt.getSource();
            selectedParticipant = cb.getSelectedIndex();
        }
    });
    participantLabel = new JLabel();
    eventList = new JComboBox<String>(new String[0]);
    eventList.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            JComboBox<String> cb = (JComboBox<String>) evt.getSource();
            selectedEvent = cb.getSelectedIndex();
        }
    });
    registerButton.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            registerButtonActionPerformed();
        }
    });
    addParticipantButton.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            addParticipantButtonActionPerformed();
        }

    });
    addEventButton.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            addEventButtonActionPerformed();
        }
    });
}
```

3. Finally, open the `EventRegistration` class in the `ca.mcgill.ecse321.eventregistration.application` package, and run the Java application with the newly created graphical interface:

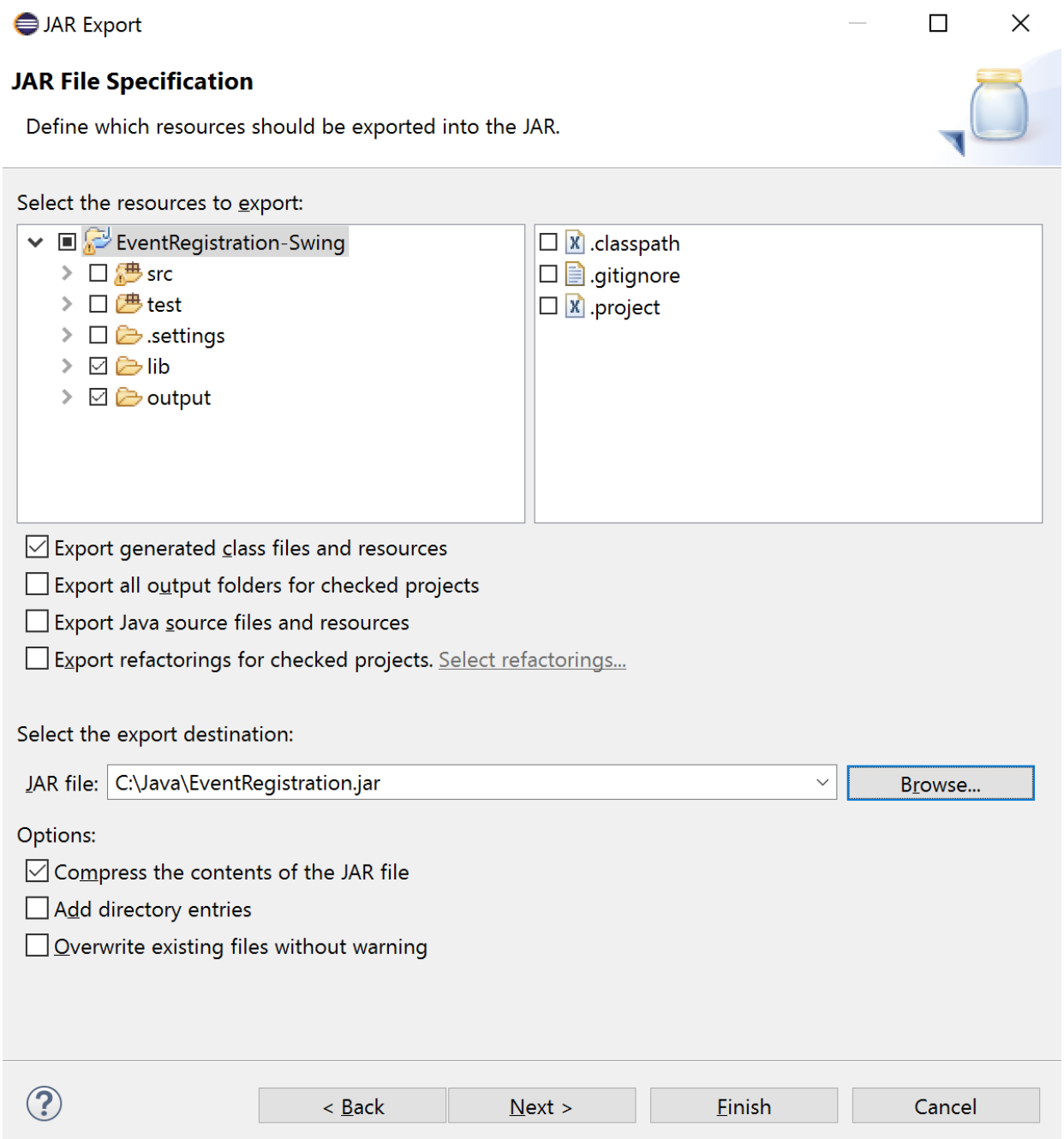
```
public static void main(String[] args) {
    final RegistrationManager rm =
PersistenceXStream.initializeModelManager(fileName);

    // start UI
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new EventRegistrationPage(rm).setVisible(true);
        }
    });

}
```

This completes the tutorial of the Event Registration Desktop application.

Note: To submit your assignment you need to create a zip file by right clicking on the project and **Export** | **General** | **Archive File**

1.10. Exporting the Java project

1. Right click on the project folder, then **Export...**
2. Select **Java** | **JAR file** from the dialog and click **Next**
3. Exclude the following project-specific files in the rigth pane (if they exist)
    - `.classpath`
    - `.gitignore`
    - `.project`
4. Exclude the `src` and `test` folders in the left pane
    - Ensure that **.class** files are exported (see the check box)
    - Ensure that the **lib** folder is included



5. Select a location where the **.jar** file will be generated

6. Click on **Finish**
7. Ignore the warnings by click on **OK**

2. Android